$Id: asg5-intermed-lang.mm,v 1.25 2018-11-16 12:13:20-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps104a-wm/Assignments
URL: http://www2.ucsc.edu/courses/cmps104a-wm/:/Assignments/

1. Overview

An intermediate language is a very low level language used by a compiler to perform optimizations and other changes before emitting final assembly language for some particular machine. It generally matches common assembly language statement semantics, but in a typeful manner.

SYNOPSIS

`oc` [`-ly`] [`-@` *flag* ...] [`-D` *string*] *program*`.oc`

All of the requirements for all previous projects are included in this project. For any input file called *program*`.oc` emit an intermediate language file called *program*`.oil`, which can then be compiled into assembly language using `gcc`.

program	→	[structdef]... [stringdef]... [vardef]... [function]...
structdef	→	'`struct`' *TYPEID* '`{`' [type *IDENT* '`;`']... '`}`' '`;`'
stringdef	→	'`char*`' *IDENT* '`=`' *STRINGCON* '`;`'
vardef	→	type *IDENT* '`;`'
function	→	type *IDENT* '`(`' parameters '`)`' fnbody
fnbody	→	'`{`' [statement]... '`}`' | '`;`'
parameters	→	type *IDENT* ['`,`' type *IDENT*]... | '`void`'
statement	→	*LABEL* '`:;`'
	→	['`if`' '`(`' ['`!`'] operand '`)`'] '`goto`' *LABEL* '`;`'
	→	'`return`' [operand] '`;`'
	→	[type] *IDENT* '`=`' expression '`;`'
	→	'`*`'*IDENT* '`=`' expression '`;`'
	→	[type *IDENT* '`=`'] *IDENT* '`(`' [operand ['`,`' operand]...] '`)`' '`;`'
expression	→	operand binop operand | unop operand | selection | operand
binop	→	'`+`' | '`-`' | '`*`' | '`/`' | '`%`' | '`==`' | '`!=`' | '`<`' | '`<=`' | '`>`' | '`>=`'
unop	→	'`+`' | '`-`' | '`!`' | '`*`'
selection	→	'`&`' *IDENT* '`[`' operand '`]`' | '`&`' *IDENT* '`->`' *IDENT*
operand	→	*IDENT* | *INTCON* | *CHARCON* | '`sizeof`' '`(`' type '`)`'
type	→	'`void`' | '`void*`' | '`char*`' | '`char**`' | '`char***`'
	→	'`int`' | '`int*`' | '`int**`' | '`struct`' *TYPEID*'`*`'
	→	'`struct`' *TYPEID*'`**`' '`struct`' *TYPEID*'`***`'

Figure 1. Grammar of `oil`

`int` *IDENT*	`int` *IDENT*	`int[]` *IDENT*	`int*` *IDENT*
`string` *IDENT*	`char*` *IDENT*	`string[]` *IDENT*	`char**` *IDENT*
TYPEID IDENT	`struct` *TYPEID*`*` *IDENT*	*TYPEID*`[]` *IDENT*	`struct` *TYPEID*`**` *IDENT*

Figure 2. Type declarations in `oc` and `oil`

2. Intermediate Language

The intermediate language chosen here looks very much like C, except that, for the most part, each `oil` statement should be capable of translating into a single assembly language instruction, or only a few. Unlike most assembly languages, `oil` is typed as to basic data types, namely char, int, and pointer. The size of the int and pointer types are dependent on the underlying architecture. The char type is only used in strings.

The basic grammar of `oil` is given in Figure 1, and uses the same metanotation as was used in the definition of `oc`. Figure 2 shows the relationship between types in `oc` and the corresponding C-compatible types in `oil`. The extra types listed in Figure 1 but not in Figure 2 are used only when the address operator (`&`) is applied to take the address of a field selection, which is then dereferenced when applied. The intermediate language has two datatypes : integers and pointers. Characters only appear in integers and strings.

2.1 Program and Function Structure

An `oil` program is structured in a way similar to C, except that it looks more like assembly language. All code is either aligned with the left margin or indented by 8 spaces.

(a) Structure definitions come first, with the `struct` keyword and closing brace left-aligned, and the fields indented, as in :

```
struct foo {
        char** foo_string_array;
        struct foo* f_foo_pointer;
};
```

(b) Then all string constants are listed in the order they appear in the program. They are unlikely to be able to be used as immediate operands in assembly language. Global declarations are all left-aligned, as in :

```
char* s1 = "Hello, world!\n";
```

(c) Then all global variable declarations that appeared in the program, but without initialization, because in C, static initialization is done at compile time.

(d) Finally, functions are emitted with one parameter per line. The function name and the two braces are left aligned, as are label statements, but everything else is indented, as in :

```
int add (
        int _1_a,
        int _1_b)
{
        int i2 = _1_a + _1_b;
        return i2;
}
```

2.2 Statements

Statements are all indented, except for labels, and all declarations of local variables must be initialized. Expressions other than calls are restricted to three-address instructions, with a destination and at most two sources.

(a) A label is emitted unindented on a line by itself. It consists of a keyword from the source language followed by the coordinates taken from the node.

```
while_4_20_9:;
```

(b) A `goto` statement may be conditional or unconditional, and if conditional is preceded by an `if` and an operand (possibly complemented) in parentheses.

```
goto while_4_20_9;
if (!b36) goto while_4_29_9;
```

(c) A `return` statement may or may not have an operand, depending on the requirements of the original `oc` program.

```
return _5_foobar;
return;
```

(d) An assignment statement takes an expression as an operand. A type is present if the identifier has not previously been declared, and absent if it has.

```
int i5 = _2_n + 45;
_2_n = i5;
char* a6 = &_3_a[i6];
*a6 = i7
```

(e) A function call may only have operands in parentheses, but may have as many as appropriate. A void function has no assignment on the left. All other functions must have it, and the result is placed into a temporary.

2.3 Expressions

Expressions are simple and restricted to three-address code.

(a) Binary operators are the same as they are in `oc` and have already been type checked. As in C, 0 means false and anything else means true. The unary operators are also the same.

(b) Address computation for indexing and field selection is taken care of by the back end, alghough a type checker would normally assign offsets of fields in structures and offsets of local variables from the frame pointer.

(i) The type of a subscript address is the same as the type of the array. So if we have `char** _3_a`, then a subscripting operation might be :

```
char** a5 = &_3_a[_3_i];
```

and when it is used, `a5` must be dereferenced, as in :

```
_3_n = *a5;
*a5 = _3_n;
```

depending on whether the dereference is an lvalue or an rvalue.

(ii) The type of a selection has one more level of pointer than the declaration of a field. So if a `struct` has a field `char** foo_bar;`, then the selection might be :

```
char*** a9 = &_3_ptr->foo_bar;
```

(c) Figure 2 show the types in `oc` and their corresponding types in `oil`. Note that the asterisk representing a pointer cuddles up against the following identifier, not against the type.

Category	Format string	Format arguments	Example
Global names	`"%s"`	identifier	`global_var`
Local names	`"_%d_%s"`	block, identifier	`_3_ix`
Stmt labels	`"%s_%d_%d_%d"`	keyword, file, line, offset	`if_4_21_9`
Structure typeids	`"%s"`	structure_typeid	`foo`
Field names	`"%s_%s"`	structure_typeid, field_name	`foo_bar`
Virtual registers	`"%c%d"`	register_category, number	`i67`

Figure 3. Summary of name mangling

2.4 Name Mangling

Assembly language generally uses a completely flat symbol table, so names need to be mangled in order to avoid clashing global names with various local scopes and the C library to which a program is linked. Figure 3 shows the `fprintf`(3) format items and arguments to be used to mangle a name.

(a) All structure names, function names, and global variables will be presented as is, "`foo`". Global names belong to block 0.

(b) Local names will be prefixed with an underscore with a block number between the underscores, as in "`_1_n`". Local names belong to blocks whose number is greater than 0.

(c) Statement labels are emitted with a keyword from the language followed by the file number, line number, and offset from the token. The following keywords are used : "`while`", before a while expression ; "`break`", the statement to transfer to for a false test ; "`else`", to branch to if the if-test is false and there is an `else` ; and "`fi`", to skip the else part.

(d) Field names are prefixed with the name of the structure to which they belong. So a field from `struct foo { int ix; };` would be called `foo_ix`.

(e) Virtual registers are just assigned numbers in sequence throughout the entire program starting with 1, 2, 3, etc. The letter used is `i` for an `int`, `s` for a *STRINGCON* address, `p` for a direct data pointer, and `a` for an indirect (dereferenceable) address pointing at data. E.g., `i2`, `s3`, `p4`, `a5`.

(f) Indirect addresses are the result of evaluating selections, which must be dereferenced when used. All other addresses are used directly.

```
int* a6 = &_3_array[_3_i];
char** a8 = &_3_foo->f_foo_bar;
```

3. Code Emission

Code is emitted from the AST with the benefit of the the symbol table and type checker. Since the suffix `oil` is not recognized by `gcc`, in order to make it compile

```
1  #!/bin/sh -x
2  # $Id: ocl,v 1.3 2012-11-16 20:57:44-08 - - $
3  for file in $*
4  do
5     gcc -g -o $file -x c $file.oil oclib.c
6  done
```

Figure 4. `oil-examples/ocl`

the code, the **`-x`** option must be used. Figure 4 shows a shell script that will compile into an executable, given the name of the executable.

The file **`oclib.h`** is bilingual for both **`oc`** and **`gcc`**.

(a) Figure 5 shows the bilingual source file.

(b) Figure 6 show the file preprocessed for **`oc`**, as will be seen by the **`oc`** compiler when included in an **`oc`** program. Note that the **`#define`**s **`EOF`** and **`assert`** are not shown in the preprocessed figures, but will be used by the preprocessor to process an **`oc`** program.

(c) Figure 7 shows it preprocessed for **`gcc`**, with names mangled names correctly when **`__OCLIB_C__`** is defined, as is done when included from **`oclib.c`**.

Figures 8 and 9 show the code for **`oclib.c`**, the library written in C.

3.1 Top Level Description

First, a prolog is emitted, then the children of the root are traversed in their own sequence to build the various parts of the program.

(a) Then all structure definitions are traversed and output as described above, with names mangled, and fields properly indented.

(b) Next, all string constants are given names as described above. This should not require a complete tree traversal. Retrofit either the construction of the AST or the type checking module to put all *STRINGCON* AST nodes into a queue.

(c) Then all global variables, immediate children of the root, and output, with any initializen given. Initialization must be simple as per the grammar in project 3.

(d) Then all functions are output, as described below.

3.2 Emitting Functions and Statements

Function emission is done by a complete depth-first mostly post-order traversal of the AST for that function. Prototypes need not be emitted. They are used only in type checking.

(a) Output the function's return type, name, and mangled parameter names. Mangle the parameter names using the block number the function created.

(b) Following that, indented, are the declarations of all of the parameters.

(c) An opening brace, unindented, at the start of the function's body, and a closing brace, unindented, after the function is finished.

(d) A block simply has its children traversed.

(e) A **`while`** has two children. For the **`while`** and **`break`** labels, both use the serial number of the **`while`** token.
  (i) Emit : **`while_%d_%d_%d:;`** unindented.
  (ii) Emit the first child, unless it is an operand.
  (iii) Emit : **`if (!`***operand***`) goto break_%d_%d_%d;`**
  (iv) Emit the statement.
  (v) Emit : **`goto while_%d_%d_%d;`**
  (vi) Emit : **`break_%d_%d_%d:;`** unindented.

(f) For an **`if-else`** statement, do the following :
  (i) Emit the first expression, unless it is an operand.
  (ii) Emit : **`if (!`**operand**`) goto else_%d_%d_%d;`**
  (iii) Emit the second child (consequent statement).
  (iv) Emit : **`goto fi_%d_%d_%d;`**
  (v) Emit : **`else_%d_%d_%d:;`** unindented.
  (vi) Emit the third child (alternate statement).
  (vii) Emit **`fi_%d_%d_%d:;`** unindented.

(g) For an **`if`** with no **`else`**, do the following :
  (i) Emit the first expression, unless it is an operand.
  (ii) Emit : **`if (!`**operand**`) goto fi_%d_%d_%d;`**
  (iii) Emit the second child (consequent statement).
  (iv) Emit **`fi_%d_%d_%d:;`** unindented.

(h) If a **`return`** statement has an expression, emit the expression, unless it is an operand, then return the operand. If not, just **`return`**.

3.3 Emitting Expressions

Expressions are always emitted using a strict depth-first post-order traversal, accumulating values into virtual registers. This results in some redundant virtual registers, but the back end can use copy propagation to remove them during register allocation.

(a) In each case of generating code into an operand, generate a declaration of a virtual register with the appropriate letter, **`i`**, **`s`**, **`a`**, or **`p`**, and assign it the value of the expression thus evaluated.

(b) The binary operators are the same in both **`oc`** and **`oil`**, and after emitting an instruction, record the register number in the AST node itself. This will require adding yet another field. Similarly, handle the unary operators. Example :

```
int i5 = _3_n + 1;
```

(c) An allocator uses a call to **`xcalloc()`** to allocate the storage, based on the type of allocator, for some structure type T and pointer number i :

(i) '`new`' *TYPEID* :
Emit : `struct` T`* p`i `= xcalloc (1, sizeof (struct` T`));`

(ii) '`new`' '`string`' '`(`' expression '`)`' :
Emit the expression, unless it is an operand.
Emit : `char* p`i `= xcalloc (`$opnd$`, sizeof (char));`

(iii) '`new`' basetype '`[`' expression '`]`' :
Emit the expression, unless it is an operand.
Emit one of the following, depending on the *basetype* :

- `char* p`i `= xcalloc (`$opnd$`, sizeof (char));`
- `int* p`i `= xcalloc (`$opnd$`, sizeof (int));`
- `char** p`i `= xcalloc (`$opnd$`, sizeof (char*));`
- `struct` T`** p`i `= xcalloc (`$opnd$`, sizeof (struct` T`*));`

(d) For a function call, evaluate each of the arguments into registers, and generate the call instruction. A `void` function is just called. A non-`void` function has its result captured into a register.

(e) An *INTCON* may be emitted as a decimal, octal, or hexadecimal number, as in C. A *CHARCON* is emitted as is. The constants `null` and `false` are emitted as the literal `0`, and `true` is emitted as the literal `1`.

(f) Constants (except strings) and variables are never captured in registers, but instead are used as immediate operands.

4. An Example

Figure 10 contains a sample `oc` program. Figure 11 contains the output file generated from it, except that the code generated from the `#include` header file has been omitted. It is assumed that `cpp` here has recognized this file as the fifth file in sequence and that the header file has generated 11 blocks so that the code presented is the 12th block.

```
 1  // $Id: oclib.h,v 1.20 2018-11-08 14:39:15-08 - - $
 2
 3  #ifndef __OCLIB_H__
 4  #define __OCLIB_H__
 5
 6  #ifdef __OCLIB_C__
 7
 8  typedef char* cstring;
 9  void* xcalloc (int nelem, int size);
10  void putint (int);
11  void putstr (const cstring);
12  cstring getword (void);
13  cstring getln (void);
14  void endl();
15
16  #else
17
18  #define EXIT_SUCCESS 0
19  #define EXIT_FAILURE 1
20  #define EOF (-1)
21  #define char int
22  #define bool int
23  #define true 1
24  #define false 0
25  #define assert(expr) { \
26             if (not (expr)) { \
27                __assert_fail (#expr, __FILE__, __LINE__, __FUNC__); \
28             } \
29          }
30  void __assert_fail (string expr, string file, int line, string func);
31  void putchar (char c);
32  void putint (int i);
33  void putstr (string s);
34  int getchar();
35  string getword();
36  string getln();
37  void endl();
38  void exit (int status);
39
40  #endif
41
42  #endif
43
```

Figure 5. `oc-programs/oclib.h`

```
 1  # 1 "oc-programs/oclib.h"
 2  # 1 "<built-in>"
 3  # 1 "<command-line>"
 4  # 31 "<command-line>"
 5  # 1 "/usr/include/stdc-predef.h" 1 3 4
 6  # 32 "<command-line>" 2
 7  # 1 "oc-programs/oclib.h"
 8  # 30 "oc-programs/oclib.h"
 9  void __assert_fail (string expr, string file, int line, string func);
10  void putchar (int c);
11  void putint (int i);
12  void putstr (string s);
13  int getchar();
14  string getword();
15  string getln();
16  void endl();
17  void exit (int status);
```

Figure 6. `cpp oc-programs/oclib.h`

```
 1  # 1 "oc-programs/oclib.h"
 2  # 1 "<built-in>"
 3  # 1 "<command-line>"
 4  # 1 "/usr/include/stdc-predef.h" 1 3 4
 5  # 1 "<command-line>" 2
 6  # 1 "oc-programs/oclib.h"
14  typedef char* cstring;
15  void* xcalloc (int nelem, int size);
16  void putint (int);
17  void putstr (const cstring);
18  cstring getword (void);
19  cstring getln (void);
20  void endl();
```

Figure 7. `cpp -D__OCLIB_C__ oc-programs/oclib.h`

```
 1  // $Id: oclib.c,v 1.96 2018-11-26 13:03:09-08 - - $
 2
 3  #include <assert.h>
 4  #include <ctype.h>
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include <string.h>
 8
 9  #define __OCLIB_C__
10  #include "oclib.h"
11
12  typedef int (*ctype_fn) (int);
13  cstring scan (ctype_fn skip_over, ctype_fn stop_at) {
14     int byte = 0;
15     do {
16        byte = getchar();
17        if (byte == EOF) return NULL;
18     } while (skip_over != NULL && skip_over (byte));
19     size_t buf_size = 16;
20     cstring buffer = malloc (buf_size);
21     assert (buffer != NULL);
22     size_t end_pos = 0;
23     do {
24        buffer[end_pos++] = byte;
25        if (end_pos >= buf_size) {
26           buf_size *= 2;
27           buffer = realloc (buffer, buf_size);
28           assert (buffer != NULL);
29        }
30        buffer[end_pos] = '\0';
31        byte = getchar();
32     }while (byte != EOF && !stop_at (byte));
33     return buffer;
34  }
```

Figure 8. `oc-programs/oclib.c`, part 1

```
35  ^L
36
37  int isnl (int byte)           { return byte == '\n'; }
38  void putint (int val)         { printf ("%d", val); }
39  void putstr (const cstring s) { printf ("%s", s); }
40  cstring getword (void)        { return scan (isspace, isspace); }
41  cstring getln (void)          { return scan (NULL, isnl); }
42  void endl()                   { putchar ('\n'); }
43
44  void* xcalloc (int nelem, int size) {
45     void* result = calloc (nelem, size);
46     assert (result != NULL);
47     return result;
48  }
49
```

Figure 9. **`oc-programs/oclib.c`, part 2**

```
 1  #include "oclib.h"
 2  int fac (int n) {
 3     int f = 1;
 4     while (n > 1) {
 5        f = f * n;
 6        n = n - 1;
 7     }
 8     return f;
 9  }
 9  int main() {
10     int n = 1;
11     while (n <= 5) {
12        puti (fac (n));
13        endl();
14        n = n + 1;
15     }
16     return 0;
17  }
```

Figure 10. Sample program **`fac.oc`**

```
 2  int fac (
 3          int _12_n)
 4  {
 5          int _12_f = 1;
 6  while_5_2_3:;
 7          char b1 = _12_n > 1;
 8          if (!b1) goto break_5_2_3;
 9          int i1 = _12_f * _12_n;
10          _12_f = i1;
11          int i2 = _12_n - 1;
12          _12_n = i2;
13          goto while_5_2_3;
14  break_5_2_3:
15          return _12_f
16  }
17  int main (void)
18  {
19          int _2_n = 1;
20  while_5_11_0:;
21          char b2 = __n <= 5;
22          if (!b2) goto break_5_11_0;
23          int i3 = __fac (__n);
24          __puti (i3);
25          __endl();
26          int i4 = __n + 1;
27          __n = i4;
28          goto while_5_11_0;
29  break_5_11_0:;
29          return 0;
30  }
```

Figure 11. **`Sample generated fac.oil`**